Hardware topology management in MPI applications through hierarchical communicators

Brice Goglin

Inria, LaBRI, Univ. Bordeaux, CNRS, Bordeaux-INP
200 Avenue de la Vieille Tour
33405 Talence – France

Emmanuel Jeannot

Inria, LaBRI, Univ. Bordeaux, CNRS, Bordeaux-INP
200 Avenue de la Vieille Tour
33405 Talence – France

Farouk Mansouri

DDN Storage
10 rue Andras Beck,
92360 Meudon La forêt France

Guillaume Mercier

Inria, LaBRI, Univ. Bordeaux, CNRS, Bordeaux-INP
200 Avenue de la Vieille Tour
33405 Talence – France

Abstract

The MPI standard is a major contribution in the landscape of parallel programming. Since its inception in the mid 90s it has ensured portability and performance for parallel applications on a wide spectrum of machines and architectures. With the advent of multicore machines, understanding and taking into account the underlying physical topology and memory hierarchy have become of paramount importance. On the other hand, providing abstract mechanisms to manipulate the hardware topology is also fundamental. The MPI standard in its current state, however, and despite recent evolutions is still unable to offer mechanisms to achieve this. In this paper, we detail several additions to the standard for building new MPI communicators corresponding to hardware hierarchy levels. It provides the user with tools to address hardware topology and locality issues while improving application performance.

Keywords: Hierarchy, Hardware Topology, Message Passing

Email addresses: `brice.goglin@inria.fr` (Brice Goglin), `emmanuel.jeannot@inria.fr` (Emmanuel Jeannot), `fmansouri@ddn.com` (Farouk Mansouri), `guillaume.mercier@bordeaux-inp.fr` (Guillaume Mercier)

1. Introduction

Parallelizing or writing from scratch a parallel application is a very challenging task and this challenge has become even more important due to the current trend in processors design and supercomputers architecture. Indeed, the hardware that the programmer has to tackle becomes more and more hierarchically organized. For instance, CPUs now feature various levels of memories that have different properties in terms of size, performance and even nature. As a consequence, a parallel application performance is likely to be impacted by the communication occurring between processes and by the way they access data. Thus, if a process accesses some data located in a memory bank physically far from the core it currently executes on, a penalty shall occur (NUMA effect). Also, if two processes share a cache level, they will communicate one with the other more efficiently. This is known as the *data locality issue*. Indeed, a recent survey on MPI usage for future exascale systems [1] outlines that a significant fraction of applications (18 out of 48) exhibit sensitivity to the hardware topology.

In order to better exploit the underlying hardware, applications need to take this locality phenomena into account. On one hand, they need to get a better grasp of the underlying physical architecture they are running on. The current success of `hwloc` [2] demonstrates the need for such a tool and the information it allows to gather. This knowledge may offer the possibility of optimize the code, to better exploit the memory hierarchy or the network topology as well as a global and comprehensive view of the hardware (a continuum of hierarchies between the network and the memory). However, on the other hand, it is required, for performance portability reasons and because domain scientists and application programmers are not hardware topology specialists, that the proposed scheme to take locality into account is done through high-level, portable abstractions.

Having efficient yet abstract mechanisms to deal with this issue is a difficult task. To this end, a relevant programming model (along with tools and libraries that implement it) can be of valuable help. The most obvious and natural choice would be to first look at what current parallel programming standards and libraries offer in this area.

In the case of the Message Passing Interface (MPI) [3], some mechanisms are already available that could make it possible for an application to exploit the hardware hierarchy to improve interprocess communication and locality. For instance, in the case of virtual topology management routines, such as `MPI_Dist_graph_create`, the `reorder` argument can be used to create a topology where processes are reorganized according to the underlying physical topology of the target architecture.

However, there are several issues with this approach: first, this argument *might* be used in this fashion, but that is not necessarily the case, which means that the expected behaviour is totally implementation-dependent (and thus not standard) hence likely to change from one implementation to the other or even worse from one implementation version to the other. An application cannot rely on a particular MPI implementation version to be able to use the specific features it needs. Then, in absence of dedicated and relevant mechanisms directly within the standard, an application is forced to use some side-effects of already available features, which constitutes an issue in terms of both interface expressiveness and usability.

In this paper, we present abstraction mechanisms that can help programmers to structure their applications based on physical topology criteria. Such structure can then be used to improve data locality or communication performance by taking into account information that would be otherwise unavailable to the underlying MPI implementation. We also present the implementation of this abstraction and mechanisms in the context of the MPI standard.

This paper is organized as follows: the current status of the MPI interface with regard to hardware topology management is discussed in Section 2 and our proposal is detailed in Section 3. Its implementation is described in Section 4 and examples of uses are detailed in Section 5. Section 6 describes the target applications of this work and experimental results are analyzed in Section 7, while Section 8 concludes this paper and give potential future directions.

2. Hardware topology management and the MPI Standard

In this section, we shall examine the current possibilities offered by the MPI standard to deal with the issue of hardware topology management in parallel applications. One key-characteristic of the MPI standard is its hardware-agnosticism. Indeed, it makes no assumptions about the hardware on which the application is going to be deployed and run. This behaviour ensures the portability of parallel programs using MPI. It is to be noted also that despite this independence from any hardware considerations in its programming model, the MPI standard and programming model does not prevent from accessing the hardware topology directly from an application.

2.1. Interactions with external tools

One way to tackle this issue is precisely to use an altogether different tool or interface, fully external to MPI. That is, one current practice is to deal with tools representing the hardware topology such as `hwloc` [2], LibNuma [4], or Pthread sched [5]. These libraries could give a relevant representation of the hardware structure and components. However, they are rather low-level tools and need a good knowledge of the underlying hardware to be used correctly and efficiently. In addition, this practice increases the complexity of developing and supporting codes. Last, as these tools or interfaces are not standard, portability is not guaranteed.

2.2. Current status in the MPI standard and its implementations

The MPI library even offers some means to better understand the nature of the physical architecture in order to exploit it to its full potential. For instance, the extension of Remote Memory Operations with Shared Memory operations in the standard has allowed programmers to structure their applications to take into account the fact that processes are located on the same machine. This can be seen as an alternative to the use of multithreading when memory consumption is at stake [6, Chapter 16]. In this particular case, the MPI standard acknowledges that some physical resource (e.g. memory) is shareable between processes and offers the tools to actually access this resource.

However, despite the presence of some mechanisms in MPI to better understand and use the hardware, they address the issue only partially. Some MPI *implementations* offer mechanisms but since they are implementation-dependent they are typically non-standard. Portability is therefore not guaranteed. The bottom line is: an application should not (or cannot) rely on a specific *implementation* nor on a specific *version* of an implementation (which is even worse). On the other hand, improving performances and scalability of applications is more efficient when locality is exposed during their design steps rather than only relying on MPI implementation optimizations. Thus, the MPI programming model as specified by the standard needs to offer high- level abstractions to take into consideration architecture topology at an early stage and before calling implementations features. The following paragraphs describe some of these mechanisms and their shortcomings.

2.2.1. Shared Memory constructs

`MPI_Comm_split_type` can accept `MPI_COMM_TYPE_SHARED` as a value for its `split_type` argument. The outcome is a communicator that encompasses all processes in the original communicator that can create a shared memory region[1] By using the communicators produced by this function, it is possible to structure an application in order to take into account the fact that memory is shared: for instance direct memory accesses are possible instead of relying on message-passing exchanges handled by the MPI library. The overheads of the library in case of shared-memory are eliminated. However, the various levels of cache, a major part of the hardware hierarchy, are left unexploited. As explained previously, this memory hierarchy is becoming more and more complex and performance gains are expected from a relevant exploitation of it.

2.2.2. Process topologies and reordering

The various process topologies available in MPI can help to structure an application but they are virtual topologies and quoting the standard itself: "The virtual topology can be exploited by the system in the assignment of processes to physical processor, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI." Some implementations use the `reorder` parameter of some functions (e.g. `MPI_Dist_graph_create` [7], etc.) and take the opportunity to retrieve hardware information and make use of it [8, 9, 10]. Some others tailor the topology routine of MPI to a specific hardware [11, 12]. This, of course, falls into the non-standard category as it is implementation-dependent and is a side effect of the function. It is not the primary goal of it and the fact that the underlying hardware is efficiently exploited can be seen as a bonus.

2.2.3. MPI Sessions

MPI sessions are a new concept that is currently being discussed by the MPI Forum. In its current state, the standard only allows to call `MPI_Init` and `MPI_Finalize` a single time in an application. This can raise some issues when multiple libraries that internally rely on MPI are used concurrently. Sessions can be seen as a lightweight construct, even lighter than groups. A session encompasses MPI processes and some information can be attached to it, for instance about the application or the hardware. In this case, the sharing of hardware resources could be exploited by the application with several different MPI sessions.

2.2.4. Process Managers and process mapping

One last critical point regards process managers. They can also be of help when it comes to exploit the underlying hardware. Indeed, through their process mapping and binding options, they can allow the user to finely control the way the various MPI application processes are dispatched and executed [13]. Thanks to an adequate placement policy enforced by both these mapping and binding parameters, it is possible to take into account the physical topology and reduce the communication costs for instance [14, 15]. This is also used to improve collective communication performance [16], Unfortunately, these options are totally non-standard and can even change from one version of a process manager to the other.

Even though we consider this specific point to be outside the scope of this paper, it is strongly tied to the issue of hardware topology management in MPI applications. Indeed, in the absence of a

[1] It is obviously the case when they are located on the same machine, but could also be the case if the processes are on different machines linked by a network *à la* SCI.

(relevant) mapping/binding of processes, the performance improvements of taking into account the underlying hardware topology are not expected to be as high as if an efficient policy is enforced. For the rest of this paper, we make the assumption that the user knows the consequences of adopting a relevant mapping/binding process policy and chooses one accordingly.

3. Proposed Extensions to the MPI standard

As exposed in the previous section, there are currently no means in the MPI standard to portably take into account the hardware topology at the application level. We think that it is important/necessary to offer high-level abstractions helping programmers to take care of locality and communication optimizations when they design their applications. This way we anticipate and facilitate an implementation work that optimizes communications according to the target architecture. Therefore, we propose to extend the MPI standard and detail in the rest of this section the relevant mechanisms and features needed to achieve these objectives.

3.1. Guidelines

Since its first version in 1994, the MPI standard has grown steadily in terms of number of available routines and functionalities. We therefore advocate for a minimal amount of changes and prefer to leverage existing mechanisms. We prefer not introducing new functions unless it is unavoidable and rather expand existing mechanisms. MPI is about communications and how they are managed. In this regard, exploiting the underlying topology boils down to be able to organize the various MPI application processes in a way that is both topologically-wise and performance-wise sensible. The same idea has already been used in the case of reordering: the processes that communicate a lot should be bound on two cores physically close to each other. Consequently, the sharing of caches is likely to decrease communication times and improve overall application performance.

The key idea is therefore to group processes into entities where a specific kind of resource is shared by all groups[2] members (i.e. processes). MPI features a concept/construct that perfectly matches our needs: the *communicator*. As a consequence, we propose to make hardware topology information and structure available at the application level through a hierarchy of communicators. We want to help an application developer and guide him/her to build the most relevant communicator (hardware) topologically-wise, without any deep knowledge of the underlying architecture and regardless of the way the application processes are mapped and/or bound on the machine.

In this hierarchy, each communicator corresponds to a specific resource that is shared by all the processes belonging to it. For instance, if a process shares a L2 cache and a L3 cache with other processes, it will be part of the communicator encompassing all processes sharing the same L2 and part of the communicator encompassing all the processes sharing this level 3 cache. Creating communicators also allows the use of collective communications among processes that share a resource. Collective communication operations are a major feature of MPI. It gives the programmer some ability to structure his/her application and encourages him/her to improve the locality factor of the communications.

[2] "*group*" has to be taken in the generic sense, we do not deal with the concept of MPI groups here.

3.2. Communicators creation

There exists a couple of functions in the MPI standard that create communicators, and one in particular if of interest for our purpose: `MPI_Comm_split`. Since the idea supporting our proposal is to create communicators based on the sharing of common hardware resources, splitting some input communicator (`MPI_COMM_WORLD` being obviously a relevant but not a mandatory candidate) is a natural fit for the goal we want to achieve: indeed the information about the sharing of the resource can be conveyed by the `color` argument. However, the outcome is not likely to be the one expected: let us take for example the case of processes mapped onto different physical nodes (for the sake of simplicity, each core of each node executes its own process). Let us then assume that each node features several L3 caches. If we want to create as many communicators as the number of L3 caches in our configuration, we cannot provide a single color value, otherwise, *all* processes would end up belonging to the same communicator. Ideally, we would like to provide the same color value, but this value has to carry a different meaning in different processes.

3.2.1. `MPI_Comm_split_type` *extension*

Fortunately, this is exactly the behaviour of `MPI_Comm_split_type`. Indeed, this function partitions the group associated with the input communicator into disjoint subgroups, based on the type specified by the value assigned to the `split_type` parameter. A single value is currently defined in MPI 3.1: `MPI_COMM_TYPE_SHARED`. When this value is used, the input communicator is split into communicators, where each new communicator represents a shared-memory domain. That is, two processes belonging to the same subcommunicator are able to create a mutually accessible shared-memory region. Obviously, there is no overlap between these new communicators. This function, along with the particular `MPI_COMM_TYPE_SHARED` value for the `split_type` parameter, already allows the user to better understand the way the processes are mapped onto the underlying hardware. It also gives the opportunity to take advantage of it since a different programming model (MPI-3 Shared Memory style) can be used in each new communicator and classical Message Passing between them. The MPI standard stipulates that implementations may define their own values for the `split_type` parameter, in order to enforce specific behaviours. The flexibility granted by this approach is counterbalanced by its sheer lack of portability. As a consequence, we propose to enrich the set of possible values for the `split_type` argument by adding a new one (`MPI_COMM_TYPE_PHYSICAL_TOPOLOGY` for instance[3]). The `info` argument can be used to pass hints to the implementation about the way the split operation should be done.

3.2.2. Properties of the hierarchical communicators

A call to `MPI_Comm_split_type` with this new value shall yield a communicator corresponding to the *lowest possible level* in the hierarchy tree representing the hardware topology. This newly produced communicator can then be used as an input argument in subsequent calls to `MPI_Comm_split_type` to produce other "child" subcommunicators that correspond to deeper levels (complete examples are detailed in Section 5). Also, the newly produced communicators should retain the following properties:

- The last valid communicator produced in this fashion may be identical to `MPI_COMM_SELF`, but not necessarily.

[3] Or any suitable and meaningful name.

- Each recursively created new communicator should be a strict subset of its parent (input) communicator. That is, a call to

  `MPI_Comm_compare(oldcomm,newcomm)`

  must return `MPI_UNEQUAL`. This property ensures that no unnecessary new communicators are created in case of redundancies of levels in the hardware topology. For instance, if a L3 cache and a L2 cache are shared between all processes, there is no need to create a communicator for both resources.

- If no valid communicator is to be created, `MPI_COMM_NULL` should (obviously) be returned.

These communicators calls will form a kind of hierarchy, mimicking the hardware one, as all new communicators are encompassed (so to speak) in their parent communicator. That is, if a process belongs to the communicator corresponding to the n-th level of the hierarchy, it also belongs to all communicators corresponding to levels 0 to $n-1$. It is important to note that our abstraction does not make any distinction between the network hierarchy and the nodes internal memory hierarchy. We provide means to organize an application according to the *structure* of the hardware, not its *nature*.

3.2.3. Creation of Roots Communicators

One other useful addition is the ability to create at the same time at each level of the hierarchy yet another communicator which includes all root processes of a hierarchical communicator. This forms another kind of hierarchy of its own and could ease the communication between all the levels of the hierarchy. This function could have the following prototype:

```
int MPI_Comm_hsplit_with_roots(MPI_Comm oldcomm,
                               MPI_Info info,
                               MPI_Comm *newcomm,
                               MPI_Comm *rootscomm)
```

With:

IN `oldcomm`: communicator (handle)

IN `info`:info argument (handle)

OUT `newcomm`: communicator (handle)

OUT `rootscomm`: communicator (handle)

`newcomm` is the same communicator created by a call to `MPI_Comm_split_type` with `MPI_COMM_TYPE_PHYSICAL_TOPOLOGY` as value for the `split_type` parameter. `rootscomm` is the communicator containing all processes that are roots in `newcomm`. A valid roots communicator can only be returned if the root process of `oldcomm` calls this function. MPI_COMM_NULL is otherwise returned by non-root processes. This function prototype shows that both *key* and *color/split_type* parameters are missing, as regularly found in others MPI split routines. In the case of the *color/split_type* parameter, since this routine is called to produce a hierarchy of hardware-topologically meaningful communicators, it is therefore unneeded. As for the *key* parameter, we

chose in our design to retain the rank of the calling process in the newly produced communicator. However, should this be considered not flexible enough, the interface could be modified easily, as this point is not influenced whatsoever by the hypotheses formulated on the newly created communicators.

3.2.4. Interaction with process mapping/binding policies

A special care should be taken regarding the current binding of process ranks. Indeed, the deepest level that shall be returned should correspond to the current process binding (e.g if a rank is bound to a L3 cache, no information below this level should be returned since it may use different L2 caches below when moving inside the binding). Any attempt to do so should return `MPI_COMM_NULL`. Moreover, unbound processes may move across an entire shared-memory machine and therefore cannot belong to a communicator split deeper than the "Machine" level.

3.3. Communicators characteristics query

The main aspect of our proposal deals with hierarchical communicators creation. However, we need to introduce other functions in order to make the use of these communicators more practical to the application developer. So far, with the proposed abstraction, users are able to leverage the structure of their hardware. However, more information might be needed, for instance in case of data distribution between the various communicators created at a certain level in the hierarchy.

3.3.1. Getting information for a hierarchical level

A process is able to retrieve some information about a specific communicator it belongs to with a call to the following function:

```
int MPI_Comm_get_hlevel_info(MPI_Comm comm,
                             int *num_comms,
                             int *index,
                             char **type)
```

With:

IN `comm`: communicator (handle)

OUT `num_comms`: number of siblings communicators (integer)

OUT `index`: communicator index (integer)

OUT `type`: type of communicator (string)

- `num_comms` is the number of communicators at the same level and that share the same parent communicator.
- `index` is a kind of rank for each communicator which should be contiguously numbered and starting from 0. It is the rank of the communicator among all communicators created by its parent communicator.
- `type` is a string giving information about the kind of resource that the communicator represents. It should be unambiguous, like `L2_Cache`, `L3_Cache` or `NumaNode`.

All this information should be cached by the communicator in an info object attached to it containing a set of *(key,value)* pairs properly defined when the communicator `comm` is created with a call to `MPI_Comm_split_type` or `MPI_Comm_hsplit_with_roots`. This info object creation would require the use of the `MPI_Comm_set_info` and `MPI_Comm_get_info` functions.

3.3.2. Getting the minimal level

Another helpful feature would be the ability for a programmer to know the name (type) of the *lowest level* in the hardware hierarchy that is shared by some processes. To this end, we propose to add the following function:

```
int MPI_Comm_get_min_hlevel(MPI_Comm comm,
                            int nranks,
                            int *ranks,
                            char **type)
```

With:

IN `comm`: communicator (handle)

IN `nranks`: number of MPI processes (integer)

IN `ranks`: list of MPI process ranks (array)

OUT `type`: type of the resource (string)

This function returns the name of the *lowest* level in the hierarchy shared by all the MPI processes which ranks in the communicator `comm` are listed in the `rank` array. If the calling process rank is not among the ranks listed in the array passed as an argument, the type returned should be "`Unknown`" or "`Invalid`".

3.4. Discussion

In this section, we discuss the design choices, advantages and potential drawbacks of our approach. We propose an abstraction based on existing MPI objects (the communicators), hierarchically modelling the hardware topology in order to improve performance and scalability. Communicators are often used in MPI applications and a well-understood concept to boot. A large majority of application developers are familiar with it (beyond `MPI_COMM_WORLD`). Therefore, using our proposal would require little effort, conceptually speaking. The communicators created do not feature a predetermined name, taken after the underlying resource it is supposed to represent. This ensures that there will be no need to make any change nor modifications in the future in case new levels in the hardware hierarchy should appear. This also justifies why we do not specify a maximal depth for the hierarchy. By creating "recursively" new communicators, the user is able to get all needed objects until the bottom is reached and no new communicator can be produced. The user can of course choose the desirable depth by querying the name/type of the communicator created and deciding to go further or not. This choice also explains why we do not rely on several predetermined values for the `split_type` argument [4] (e.g. one value corresponding to a specific hierarchy level) because architectural changes in hardware would require modifications to the standard.

[4] As implemented in Open MPI with the `OMPI_COMM_TYPE_*` values for instance.

It is important to note that with this proposal, it becomes possible to create *a* hierarchy of communicators corresponding to the various hardware levels, but not *the* hierarchy. Indeed, it is not mandatory that this hierarchy has to be exhaustive or complete. As a consequence, a partial implementation that cannot or do not want to expose some levels is totally acceptable. However, the various communicators must comply to the characteristics and properties described in Section 3.2.2. The only expected consequence of this lack of levels provided to the user is in terms of performance, as the users will not be able to optimize their code to the full extent of the available hardware.

Currently, our design is based on a "recursive" call to `MPI_Comm_split_type` in order to create the hierarchy. However, an alternate solution would be to introduce a function that create all communicators at once, returning an array for instance as well as its size (i.e. the hierarchy depth). Such a function is more simple to use, but compels the user to create all the levels, even undesired ones. The chosen design allows for more control at the cost of some ease of use.

One drawback of our approach is the fact that it targets architectures which are hierarchically organized. It is the case for most machines, but there are some exceptions that our model does not currently address. And the same issue arises for network topologies that are not hierarchical, such as torus for instance. In such a case, should the split be made on just a particular dimension of the torus? The `info` argument could be used to pass this information. But if some form of hierarchy can be extracted from the network topology, we can exploit it with our mechanism.

Currently, the interface proposed features one new value for the `split_type` argument of the `MPI_Comm_split_type` function and three new routines. But one open question is whether we should offer more elements. Indeed, it could be interesting to create several hierarchies of communicators, based on different criteria with different `split_type` values. For instance, making a distinction between the network topology and the nodes internal topology could be a relevant idea. The info argument in `MPI_Comm_split_type` could also be used to split a communicator directly at a desired level, without creating the whole hierarchy.

4. Implementation Details

Our proposal is implemented and available as a prototype library called *libhsplit* (for *hierarchical split*). We chose the external library route for fast prototyping reasons but our goal is to eventually integrate this work within an MPI implementation.

Our proposal is technology or software independent and implementable at the sole condition that hardware informations can be retrieved from the underlying target machine. In our case, we chose to leverage two different pieces of software: `hwloc` [2] and its extension, `netloc` [17].

4.1. An `hwloc`-based implementation

In this section, we describe the functions of the library that implements the proposal exposed previously. The first version of this library focuses only on the nodes and their internal memory hierarchy and leaves unaddressed the network hierarchy. That is, this implementation considers a flat network topology, meaning that all nodes of a cluster are connected to the same switch. The reason of such a choice stems from the software used to retrieve hardware information. Indeed, we used `hwloc` [2] to gather as much information as possible. `hwloc` is a tool that provides the user with an interface allowing to retrieve hardware details in the most portable fashion possible. `hwloc` is widely available and has already been integrated in several MPI implementations to manage hardware details. It is the case of both Open MPI and MPICH, for instance. As a consequence, this

version of our library partially captures the hardware hierarchy. However, as discussed previously in Section 3.4, this is an acceptable outcome/behaviour. Moreover, this implementation complies totally with the communicators properties listed in Section 3.2.2.

4.1.1. `MPI_Comm_split_type` *implementation*

First, we detail the implementation of the routine at the core of our proposal, that is, `MPI_Comm_split_type`. We propose to introduce a new value for the `split_type` parameter of this function. Our implementation relies on the more generic `MPI_Comm_split` operation that takes as input a `color` parameter that allows to split the input communicator into k non-overlapping subcommunicators[5]. Using this operation in our case is relevant since the subcommunicators we propose to build are non-overlapping by nature. As a consequence, the issue then boils down to determining the right color that shall be used by the regular split operation. Algorithm 1 details how this can be determined with `hwloc`:

Algorithm 1: Split Color Algorithm: cpusets refer to the way `hwloc` describes object location (set of hardware threads included in the object) or process binding (set of hardware threads where the process may run).

```
1 color ← MPI_UNDEFINED
2 obj = get_ancestor(comm); // Get the deepest obj which contains all
  processes binding
3 foreach idx← 0 .. (obj->arity) do // Find the right color
4 |  if calling_process->cpuset ⊆ obj->children[idx]->cpuset then
5 |  |  color ← idx
6 |  |  break
7 MPI_Comm_rank(comm,&rank)
8 MPI_Comm_split(comm,color,rank,newcomm)
```

- The first major thing is to determine the `hwloc` object in the hierarchy that encompasses all the processes members of the communicator that we want to split. `hwloc` conveniently proposes a routine returning the deepest object in the hierarchy in case of levels redundancy (line 2).
- Once this object has been obtained, we are able to search through its `hwloc` children objects in order to find to which specific one the calling process is bound (line 3). A match is found when the binding of the calling process is included in the considered child (line 4). These child objects are logically numbered in sequence by `hwloc`, therefore the right color is the matching child's index (line 5). If no candidate is found, the color is left to the `MPI_UNDEFINED` value which will yield an invalid (null) communicator in `MPI_Comm_split`.
- A key needs to be determined for the split. We decided to use the rank of the calling process in the input communicator (line 7).

[5] If no color is specified (`MPI_UNDEFINED`), then no valid communicator (`MPI_COMM_NULL`) is created.

- The split operation is effectively performed, with both the `color` and `key` parameters set with their values (line 8).

If the resulting communicator is valid, we then set three *(key,value)* pairs that can later be queried by a call to `MPI_Comm_get_hlevel_info` (see below). These three keys are the following:

- `MPI_COMM_HLEVEL_TYPE`: the value corresponding to this key is a name for the communicator that can help the programmers to better understand the resource they are using through the communicator. See section 3.3.1 for examples of possible string values. In our implementation, the names are provided directly by `hwloc` by the `hwloc_obj_type_snprintf` routine. By doing so, portability is enforced to some degree, or at least across MPI implementations that feature `hwloc` (which is the case of Open MPI, MPICH and MVAPICH for instance).

- `MPI_COMM_HLEVEL_NUM`: the value corresponding to this key is the total number of subcommunicators created at a specific level in the hierarchy.

- `MPI_COMM_HLEVEL_RANK`: the value corresponding to this key the "rank" of a subcommunicator among all created subcommunicators at a specific level of the hierarchy.

The last two *(key,value)* pairs can be used by programmer in case of data distribution among the various parts of the hardware architecture. These *(key,value)* pairs are then stored in the resulting new communicator with the `MPI_Comm_set_info` function. It is important to note that the keys names are not exposed to the programmer at any time. Indeed, since a query routine is provided, there is no need to expose this internal part of the library.

4.1.2. `MPI_Comm_hsplit_with_roots` *implementation*

The next function we describe is the one that returns both a subcommunicator and the communicator whose group contains all the root processes of all created subcommunicators during this step. It is possible to create this roots communicator with the current functions available in MPI. However, from a performance standpoint, an `hwloc`-based implementation (and thus a new function in the standard) is more efficient as `hwloc` gives us information that can be directly used to this purpose. It also removes the burden of implementing it by the user. That is why we proposed a single function that creates both communicators at the same time. Algorithm 2 describes how this can be performed with `hwloc`:

- The first part is similar to the algorithm explained in the previous paragraph: the subcommunicator is produced (line 1 to 8).

- In order to produce the root communicator, we need once again to find the right color to use in a second `MPI_Comm_split` operation. Only the roots of the subcommunicators will be part of the resulting communicators. Such roots are chosen as processes with rank 0 in the subcommunicators. This is a sensible choice because if a communicator is valid, i.e. not `MPI_COMM_NULL`, it shall contain at least one process and processes are numbered in sequence, starting from 0 (lines 10 and 11).

- If the calling process is determined to be a root, it will be part of the split and has to provide `MPI_Comm_split` with a color (as previously, the key used shall be its rank in the *input* communicator and not the subcommunicator). This color is chosen as the `hwloc` logical

Algorithm 2: Split Color Algorithm: variant with roots communicator creation

```
1  color ← MPI_UNDEFINED
2  obj = get_ancestor(comm); // Get the deepest obj which contains all
   processes binding
3  foreach idx← 0 .. (obj->arity) do // Find the right color
4  |  if calling_process->cpuset ⊆ obj->children[idx]->cpuset then
5  |  |  color ← idx
6  |  |  break
7  MPI_Comm_rank(comm,&rank)
8  MPI_Comm_split(comm,color,rank,newcomm)
9  color ← MPI_UNDEFINED
10 MPI_Comm_rank(newcomm,&newrank)
11 if newrank = 0 then
12 |  color ← (obj->logical_index)
13 MPI_Comm_split(comm,color,rank,rootscomm)
```

index of the *ancestor* object (line 12). Indeed, choosing a valid color value for processes with 0 rank and an invalid color otherwise will not create the intended hierarchy of roots communicators. We have to make a distinction between root processes sharing the *previous* level and this information is conveyed by the logical index of the *ancestor* object.

- The split operation is performed with the right parameters (line 13).

All the *(key,value)* pairs detailed in the previous section are also set in this function.

4.1.3. `MPI_Comm_get_hlevel_info` *implementation*

The implementation of this function is rather straightforward, as its goal is to gather the information available for a certain hierarchical level (a communicator) and pass it to the application developer in a tractable form. Since all information is stored in an MPI info object, a call to `MPI_Comm_get_info` is sufficient to retrieve the needed object. Since the various keys used (i.e. `MPI_COMM_HLEVEL_NUM`, `MPI_COMM_HLEVEL_RANK` and `MPI_COMM_HLEVEL_TYPE`) are internal to the library, we can then query for their value and passing it to the user through the arguments of the function.

We decided to not use `MPI_Comm_set_name` and `MPI_Comm_get_name` to store/retrieve the value of the `MPI_COMM_HLEVEL_TYPE` key since it is not possible to store multiple names of a more complete description of the hierarchical level the communicator represents. Also, this communicator could possibly bear a name that we do not want to replace.

4.1.4. `MPI_Comm_get_min_hlevel` *implementation*

In our current implementation, most of the strings returned by this function are generated by `hwloc`. However, some cases are not covered by `hwloc` and as a consequence, we introduced our proper strings:

- if the calling process is not part of the communicator queried upon, "Unknown" is returned

- if some processes in the list and the calling process do not belong to the same node, "System – Cluster" is returned
- if all processes in the list are local (i.e. belong to the same node), the `hwloc` name corresponding to the level they share is returned

4.2. Network Support Extension with `netloc`

The implementation described in the previous sections only addresses the node internal hierarchy, such as the memory hierarchy. However, even though this is an acceptable solution as users can exploit some (but not all) hierarchy levels, a more comprehensive solution is preferable. Indeed, the various switches in a network often form a hierarchy of their own. Allowing the users to know about it and leverage the hardware characteristics and organization of the underlying network is a compelling idea.

Our model and abstraction is able to give the user some means to optimize their code according to the *structure* of the hardware and not its *nature*. Practically, this means that a hierarchy of switches and a memory hierarchy are not seen nor treated as different things: they constitute a continuum. The issue is then to be able to retrieve the needed information about these various levels in the network, like `hwloc` is able to gather information about the memory hierarchy inside a node.

Such a tool actually exists: `netloc` [17]. `netloc` is a `hwloc` extension that specifically addresses network hierarchies and topologies. In order to create the communicators corresponding to the switches levels, we use the following experimental `netloc` routine:

```
int netloc_get_network_coords(int *nlevels,int *dims,int *coords)
```

Where:

- `nlevels` represents the number of switch levels in the network hierarchy
- `dims[i]` represents the number of switches at level `i` in the network hierarchy. `dims` is therefore an array of size `nlevels`.
- `coords[i]` represents the coordinates (or "rank") of the calling process at level `i` in the network hierarchy. `coords` is also an array of size `nlevels`.

The last level in the network hierarchy is not taken into account as it is merely a port number in the last switch. Since all nodes are connected to a switch with their own port number, this information has no use in our context and is redundant with the "Node" level.

From an implementation's point of view, adding the knowledge of the network hierarchy introduces moderate modifications to the existing code, as it was conceptually ready to support such feature. The switch level number is added as a possible string returned by the `MPI_Comm_get_min_hlevel` function.

5. Practical examples

We now detail practical examples of use of this new `split_type` value. Let us suppose that an MPI application is launched on several machines featuring the memory hierarchy depicted by Figure 1: each node is composed of two NumaNodes with a single socket (i.e. package) and 4 cores

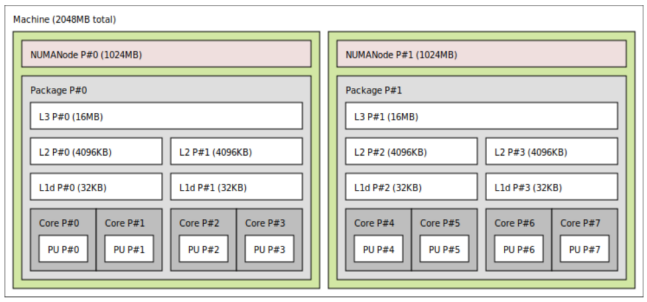

Figure 1: A hierarchical node example.

per socket. Each socket features its dedicated L3 cache and a L2 cache is shared between a pair of cores (hence 4 L2 caches in total). At some point, we assume that the various processes of the application execute the following code:

```
MPI_Comm newcomm[NLEVELS];
MPI_Comm oldcomm = MPI_COMM_WORLD;
int rank, idx = 0;
while((oldcomm != MPI_COMM_NULL) && (idx < NLEVELS)){
  MPI_Comm_rank(oldcomm,&rank);
  MPI_Comm_split_type(oldcom,
                      MPI_COMM_TYPE_PHYSICAL_TOPOLOGY,
                      rank,
                      MPI_INFO_NULL,
                      &newcomm[idx]);
  oldcomm = newcomm[idx++];
}
```

In this code snippet, `NLEVELS` is chosen appropriately so that there are enough elements in the `newcomm` array, but as we discussed previously (see Section 3.4), our proposal does not make any assumption on the total number of levels in the hardware hierarchy nor on the nature of the hardware resource the communicator is supposed to represent for the processes.

Our proposal does not feature a query function that yields that maximum number of hierarchy levels, as it would incur to an additional time the communicators hierarchy. Practically, this number is not going to be very high, a few tenths at most. As a consequence, having a statically allocated array as in our example is not an issue for memory consumption.

5.1. A simple case

In our first case, we suppose that we have an application featuring 32 processes, launched on a 4-node cluster, each node ($n_k, k \in [0..3]$) is as described previously. Let us also suppose that each process is bound to its own dedicated core. For instance, process p_x (of rank x in `MPI_COMM_WORLD`) is bound to core number i of node number k where $x = 8k + i$ ($k \in [0..3]$ and $i \in [0..7]$) [6]. When the code shown above is executed, the following communicators are created in several steps, where one step corresponds to a call to the `MPI_Comm_split_type` function:

[6] A "by node" mapping policy in conjunction with a "by core" binding policy for processes.

- **step 1:** Since processes are located on 4 different nodes, `MPI_COMM_WORLD` encompasses all these nodes. As newly created communicators have to be strictly "included" in their parent communicator, new communicators corresponding to the "Machine" level of each node shall be created. That is:

$$\texttt{newcomm}[0] = \{p_{8k}, p_{8k+1}, \ldots, p_{8k+6}, p_{8k+7}\}$$
(for each node number $k \in [0, 3]$)

However, the implementation might also decide to create an intermediate level between `MPI_COMM_WORLD` and this one to represent a level of network switches. In our case, we suppose that all nodes are connected to the same switch and no such intermediate level is created.

For the sake of simplicity, and without loss in generality, we shall now focus only on the communicators created on node number 0 in the remaining steps described below.

- **step 2:** Since all processes are now located on the same node, `newcomm[0]` encompasses the whole node. As newly created communicators have to be strictly "included" in their parent communicator, the next communicator should correspond to a level strictly smaller than the "Machine" level. The next meaningful hardware level in the target hierarchy is then the NumaNode/Package/L3 level and since the *lowest* level is chosen in case of redundancies, the "L3" level is chosen. As a consequence, two new communicators are created, one for the first L3 cache and one other for the second L3 cache. More precisely, the two newly created communicators are composed by the following processes:

$$\texttt{newcomm}[1] = \{p_{4i}, p_{4i+1}, p_{4i+2}, p_{4i+3}\}$$
(for each L3 cache number $i \in [0, 1]$)

- **step 3:** In this next step, we shall split the communicators createad during step 2, the L3 Cache communicators now are the parents communicators in the next call to `MPI_Comm_split_type`. Since there are two L2 caches/L1 caches per L3, these physical entities are the ones for which new communicators are created. Once again, the lowest level is chosen and in this second step, the new communicators correspond to the L1 caches. Practically, a new call to `MPI_Comm_split_type` with a `newcomm[1]` communicator as input will produce the following four communicators in node 0:

$$\texttt{newcomm}[2] = \{p_{2i}, p_{2i+1}\} \text{ (for each L1 Cache number } i \in [0, 3] \text{)}$$

- **step 4:** In the next step, a total of eight communicators are created, since two cores/processing units (a.k.a PU) share a single L1 cache. The "Core" level is preferred because it is the highest in the hierarchy. At this step, a total of eight communicators are created, one per process. These communicators are the following:

$$\texttt{newcomm}[3] = \{p_i\} \text{ (for each Core number } i \in [0, 7])$$

It is to be noted that if p_i called:

`MPI_Comm_compare(newcomm[3],MPI_COMM_SELF,&result)`,

then `result` would have `MPI_CONGRUENT` for value.

- **step 5:** In this last step, since we have reached the bottom of the hierarchy, no new valid communicators are produced. Therefore, all calls to `MPI_Comm_split_type` yield:

$$\texttt{newcomm}[4] = \texttt{MPI\_COMM\_NULL} \text{ (for each Core number } i \in [0, 7])$$

In the code snippet shown above, if the call to `MPI_Comm_split_type` is replaced by `MPI_Comm_hsplit_with_root`, another hierarchy of communicators is also produced, that is, the roots communicators. That is, `rootscomm[i]` is created during the same step as `newcomm[i]`.

- **step 1:** Since four communicators are created (one for each Node), the roots communicator contains four processes. Since the ranks are retained in the new communicators, this means that the root in each communicator is the process with the lowest rank. Therefore:

$$\texttt{rootscomm}[0] = \{p_{8k}\},\ k \in [0..3]$$

As in the previous case, we now examine the next step for a single node only.

- **step 2:** Since two communicators are created (one for each L3 cache), the single roots communicator contains two processes. Therefore:

$$\texttt{rootscomm}[1] = \{p_0, p_4\}$$

Please note that the root of `rootscomm[1]` is also a member of `rootscomm[0]`.

- **step 3:** Each "L3 cache" communicator is split into two "L1" subcommunicators. Hence, two roots communicators are created during this step and they contain two processes each:

$$\texttt{rootscomm}[2] = \{\mathbf{p_0}, p_2\} \text{(for L3 cache number 0)}$$
$$\texttt{rootscomm}[2] = \{\mathbf{p_4}, p_6\} \text{(for L3 cache number 1)}$$

As expected, the roots of both `rootscomm[2]` communicators are the members of `rootscomm[1]`.

- **step 4:** The same logic of step 3 is applied: each "L1 cache" communicator is split into two "Core" subcommunicators. A total of four roots communicators are then created at this step, which two processes each. That is:

$$\texttt{rootscomm}[3] = \{\mathbf{p_0}, p_1\} \text{ (for L1 cache number 0)}$$
$$\texttt{rootscomm}[3] = \{\mathbf{p_2}, p_3\} \text{ (for L1 cache number 1)}$$
$$\texttt{rootscomm}[3] = \{\mathbf{p_4}, p_5\} \text{ (for L1 cache number 2)}$$
$$\texttt{rootscomm}[3] = \{\mathbf{p_6}, p_7\} \text{ (for L1 cache number 3)}$$

Once again, the roots of `rootscomm[3]` are members of either one of the `rootscomm[2]` communicators.

- **step 5:** Since the bottom of the hierarchy is reached, no roots communicators are produced during this last step. As expected:

$$\texttt{rootscomm}[4] = \texttt{MPI\_COMM\_NULL} \text{ (8 times, one per process)}$$

5.2. A more complicated case

This second example demonstrates the flexibility of our proposal and its ability to cope with more complicated cases than the straightforward example described in the previous section. Even if we do not expect such cases to be commonplace, we emphasize that they do not raise any issue practically. In this example, we launch an application on a single 8-core node where the various processes follow their own binding policy. As a consequence, the processes are not uniformly bound on the machine. Figure 2 depicts such a case involving 8 processes in `MPI_COMM_WORLD`: processes rank 0 and 1 are bound to core 0 and 1 respectively, while processes 2 and 3 are bound on the L2 cache number 1. This means that during the application execution, these processes can execute on either core number 2 or 3, since both cores share the L2 cache number 1. Processes 4, 5, 6 and 7 are bound to NumaNode number 1. In this case they can use any of the cores that belong to this NumaNode, that is, either core 4, 5, 6 or 7. Here again, nothing prevents the core to change during the application execution.

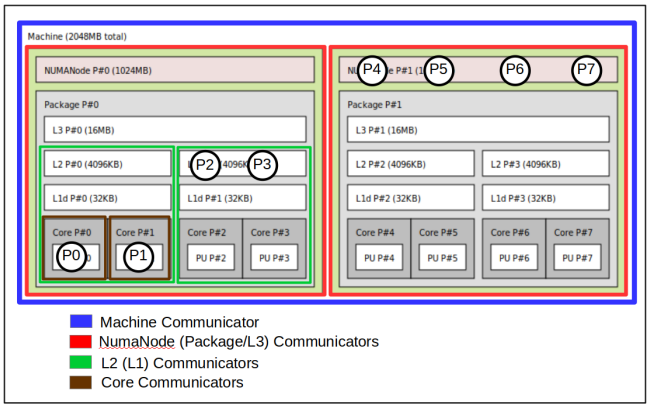

Figure 2: A case of non-uniform binding policy for processes

In this case, and according to the properties of the hierarchical communicators, we shall have the following results if the code snippet shown previously is executed by all processes:

- **step 1:** All processes are located within the same node, hence MPI_COMM_WORLD encompasses the whole node. This original communicator is the "Machine" communicator (the blue one) on Figure 2. We use this input communicator for the split operation and since the next meaningful level in the hardware hierarchy is the NumaNode/Package/L3 one, two new communicators are created, since there are two NumaNodes below the "Machine" level. As in the previous example, the *lowest* level is chosen, that is, the "L3 cache" level.

$$\texttt{newcomm[0]} = \{p_{4i}, p_{4i+1}, p_{4i+2}, p_{4i+3}\}$$
$$\text{(for L3 cache number } i \in [0, 1])$$

  These communicators are the red ones on Figure 2.

- **step 2:** In this step, the `newcomm[0]` communicators are handled differently as the processes belonging to them have different binding policies. In the case of the communicator that covers L3 cache number 0, all processes are bound to at least a L1 cache, this step is similar

to the second step of the first example. However, the case of L3 cache number 1 is very different, as the hierarchy cannot be determined since all processes can potentially move inside their L3 cache from one core to the other during the application execution, as explained in Section 3.2.4. As a consequence, no new valid communicator is created during this step for L3 cache number 1. Therefore only two subcommunicators are created:

$$\texttt{newcomm}[1] = \{p_{2i}, p_{2i+1}\} \text{ (for each L1 cache number } i \in [0,1] \text{)}$$
$$\texttt{newcomm}[1] = \texttt{MPI\_COMM\_NULL} \text{ (for each L1 cache number } i \in [2,3] \text{)}$$

These communicators are the green ones on Figure 2.

- **step 3:** From this step on, only the valid communicators can be split again, that is, the L3 cache communicators of L3 cache number 0. Since processes rank 0 and 1 are bound to cores, it is possible to create new subcommunicators that correspond to this level. However, since processes rank 2 and 3 are not bound "deeper" than the L1 cache level, no new communicator can be created. Once again, only two new valid communicators are created:

$$\texttt{newcomm}[2] = \{p_i\} \text{ (for each Core number } i \in [0,1])$$
$$\texttt{newcomm}[2] = \texttt{MPI\_COMM\_NULL} \text{ (for each Core number } i \in [2,3] \text{)}$$

These communicators are the brown ones on Figure 2.

- **step 4:** In this last step, the bottom of the hierarchy is finally reached for processes rank 0 and 1. For all other processes, the bottom has been reached in one of the previous steps. Therefore:

$$\texttt{newcomm}[3] = \texttt{MPI\_COMM\_NULL} \text{ (for each Core number } i \in [0,1])$$

Let us now examine the case where the `MPI_Comm_hsplit_with_roots` function was to be called instead of `MPI_Comm_split_type`. Nothing would change as far as the subcommunicators creation is concerned. However, fewer roots communicators would be created:

- **step 1:** In the first step, since one communicator corresponding to each L3 cache is created, the single roots communicator would contain two processes:

$$\texttt{rootscomm}[0] = \{p_0, p_4\}$$

- **step 2:** In the second step, no roots communicator is created for L3 cache number 1. The roots communicator for L3 cache 0 contains only two processes, the roots of the L2 cache subcommunicators. That is:

$$\texttt{rootscomm}[1] = \{\mathbf{p_0}, p_2\} \text{ (for L3 cache number 0)}$$
$$\texttt{rootscomm}[1] = \texttt{MPI\_COMM\_NULL} \text{ (for L3 cache number 1)}$$

Once again, we can verify that the root process of the valid `rootscomm`[1] communicator is also a member of `rootscomm`[0].

- **step 3:** In this step, only the roots communicator of the L1 cache number 0 can be created, as processes rank 2 and 3 are not bound deeper that the L1 level in the machine. Therefore:

$$\texttt{rootscomm}[2] = \{\mathbf{p_0}, p_1\} \text{ (for L1 cache number 0)}$$
$$\texttt{rootscomm}[2] = \texttt{MPI\_COMM\_NULL} \text{ (for L1 cache number 1)}$$

  The root process of `rootscomm`[2] is also a member of `rootscomm`[1].

- **step 4:** The bottom of the hierarchy is reached, two non-valid roots communicators are produced during this step, as this has already been done in previous steps for the rest of the processes in the node.

$$\texttt{rootscomm}[3] = \texttt{MPI\_COMM\_NULL}$$
$$\text{(two times, for processes 0 and 1)}$$

This demonstrates that our proposal is flexible enough to accommodate the case of non-uniform binding policies within the same MPI application. This is possible because our approach does not depend on the binding policy applied to the various processes of the application.

6. Benefits of the hierarchical model for parallel applications

In this section we propose to model the class of MPI applications that can leverage our hierarchical abstraction to enhance the performance of their communications. Let us suppose that

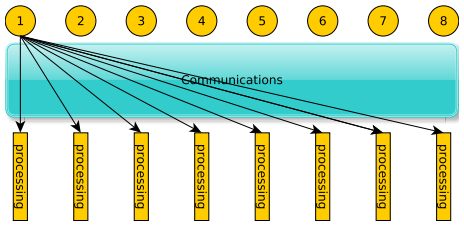

Figure 3: Superstep of MPI application family based on native collective communications

an MPI application features p MPI processes. The application execution model we focus on is in the form of a repetitive phasis of super-steps including communication operations C followed by execution operations E as shown by Figure 3. Each process needs to terminate its communication operation to start the processing of data as shown on Algorithm 3.

Algorithm 3: MPI algorithm covered by our hierarchical model

```
1 MPI_Initialization(); (// Initialisation part) foreach s← S − 1..0 do
    // Algorithm super-steps
2   MPI_Collective_Communication_Operations(); MPI_Processing_Operations();
```

If the communication operations C include collective operations, then the application can take advantage of the hierarchical model described by Algorithm 4, where L is the number of levels in the hierarchy allowed by the target architecture and generated automatically by our approach.

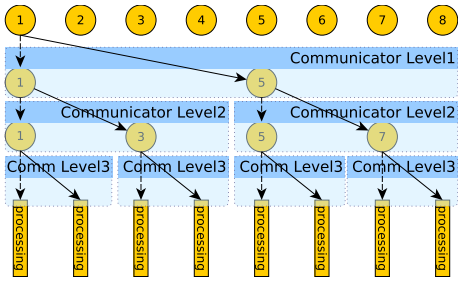

Figure 4: Superstep of MPI application family based on hierarchical collective communications

Algorithm 4: MPI algorithm based on a hierarchical communication model

```
1 MPI_Initialization();(// Initialisation part) while L ≠ MPI_Comm_NULL do
2     L ← MPI_Comm_hsplit_with_roots(); // Topological Communicators creation
3 foreach s← S − 1..0 do // Algorithm super-steps
4     foreach l← L − 1..0 do // Hierarchy levels
5         MPI_Collective_Communication_Operations(l);
6     MPI_Processing_Operations();
```

Figure 4 illustrates the application communication model based on a three-level communicator hierarchy.

Applying a hierarchical model can improve the communication times and thus the total time of the application. As a matter of fact, by exploiting more parallelism in the hierarchy levels, it is possible to enhance the scalability and performance of collective operations. In order to illustrate this claim we propose a short discussion to highlight the important features which can explain this benefit.

The collective communications are modeled with well-known works such as the Hockney [18], the LogP [19] or the PlogP [20] model. These models express the maximum time taken by a collective communication operation as a linear f unction depending on several variables such as:

- p: The number of processes performing the collective communication operation c.
- m: The size of the data exchanged between processes.
- The collective algorithm used.
- α: The hardware latency that may be a function of m and p.
- β: The hardware bandwidth that may be a function of m and p.

For example, the processing time of the simple Flat Tree broadcast is expressed with the Hockney [18] model as equal to :

$$(p-1) \times (\alpha + m\beta)$$

As shown by Algorithm 4 L levels of communications are present in the topology-aware hierarchical model according to the structure of the architecture. Thus, the time of collective operations

will be represented as a sum of L functions depending on the previous cited features but where each level is characterised by a set of specific parameters: p_i, α_i, β_i and m_i. As a consequence, the Flat Tree broadcast example shall be expressed as:

$$\sum_{i=0}^{L-1}(p_i - 1) \times (\alpha_i + m\beta_i)$$

The hardware topology-aware hierarchical model could reduce the processing time of collective operations by exploiting the following points:

- Fewer processes per level: in most applications, it is enough to use a pyramidal structure in the hierarchical model of computation. In fact, as illustrated by Figure 4, the structure based only on the roots of hierarchical communicators in upper levels is enough to perform the communication. Such a structure implies to use fewer processes p_i by level, which reduces the processing time of the collective.
- Parallelism by level: this point regards the exploitation of parallelism at each level of the hierarchy. Indeed, and except for the top level of the hierarchy, the communications inside the communicators of the same level are carried out in parallel.
- Data locality and process affinity: This point highlights the advantage of using hardware-aware communicators. Indeed, taking into account the hardware affinity of processes placement in the machine, the communication between them is enhanced thanks to the cache optimizations.
- Improved latency and bandwidth per level: this point concerns the physical latency and bandwidth which could be improved when the number of processes is small. In fact, because of the contention phenomenon, the higher the number of processes involved in the communication at the same time through an interconnect, the higher the latency and the lower the bandwidth delivered to each process.

From all the above points, collective communication operations could be improved depending on the execution conditions. For instance, if we take the Flat Tree broadcast of a single message ($m = 1$) and operated in the simple context of 8 processes and the three-level hierarchy as described by Figure 4 then the result is:

$$\text{The simple case: } Time = (8 - 1)(\alpha + \beta)$$

$$\text{The hierarchical case: } Time = (2 - 1)(\alpha_0 + \beta_0) + (2 - 1)(\alpha_1 + \beta_1) + (2 - 1)(\alpha_0 + \beta_1)$$

If we compare both expressions, it is clear that the hierarchical approach enhances the simple collective with the factor of at least (7/3) times when latencies and bandwidths are equal in both the simple and the hierarchical cases. However, in the real situation, the simple execution could generate more contention on the interconnects of each level than the hierarchical execution. Thus, the factor of enhancement of collective time could be greater than (7/3).

7. Experimental Results

In this section we present the experimentations we carried out and their results to demonstrate the benefits of our proposal. We modified two benchmarks by introducing our hierarchical model of communicators and we compared their executions on three architectures. In all cases we used the roots communicators hierarchy generated by calling the primitive described in subsection 3.2.3. In addition, we used all cores of a targeted architecture and bound an MPI process on each of them.

7.1. Platforms and architectures

For our study, we used tree architectures: a network of 10 nodes (NTW10E5) and two SMP machines (SMPE12E5, SMP20E5) from the Plafrim platform [21]. The characteristics of these architectures are given in Table 1.

Table 1: Characteristics of the used architectures. NUMA nodes are grouped by pairs based on distance in large SGI platforms. These pairs correspond to the physical blades.

Name	SMP12E5	SMP20E7	NTW10E5	NTW24E5
OS	RHEL7	SLES11	CentOS 7	Red Hat 4.8
Kernel	3.10.0	2.6.32.46	3.10.0	3.10.0
Nodes	1	1	10	24
NUMA nodes	12 (6 pairs)	20 (10 pairs)	4	4
Sockets	12	20	2	2
Cores per NUMA	8	8	6	6
Socket	E5-4620v2	E7-8837	E5-2680v3	E5-2680v3
Clock rate	2600Mhz	2660Mhz	2600Mhz	2600Mhz
Hyper-Threading	Yes	No	No	No
L1 cache	32K	32K	32K	32K
L2 cache	256K	32K	256K	256K
L3 cache	20480K	24576K	15360K	15360K
Mem Interconnect	NUMAlink6	NUMAlink5	QPI	QPI
Node Interconnect	N/A	N/A	InfiniBand	InfiniBand
Hierarchical levels	3	3	4	5
GCC	5.1	5.1	5.1	5.1
Open MPI	2.0.1	2.0.1	2.0.1	2.0.1
Hwloc	2.0-git	2.0-git	2.0-git	2.0-git

7.2. Collective Communications

The first benchmark is the broadcast and the reduce collective communication operations which we chose to test the proposed hierarchical communication model. There has been other works that aimed at exploiting hierarchy in the hardware in order to improve collective communications. In [22] a distinction is made between inter-node and intra-node communication, because shared-memory based communications are expected to be faster than their network-based counterparts. In this case, the hierarchy was limited to two levels (intra vs. inter-node) and sometimes three (intra. vs inter-cluster). We generalized this approach and make no assumption about the number of levels. Moreover, we are able to exploit the memory hierarchy inside the nodes of a cluster which is not addressed at all by these works. However, the aim here is *not* to rewrite collectives or to propose some new algorithms. Our goal is merely to experiment our abstraction in order to assess the potential gains achievable by optimizing communication and data locality.

We compared two implementations:

- *Native*: this is the Open MPI implementation of the considered collective.
- *Hierarchical*: this is a loop over the levels of the hierarchy calling the Open MPI version of the collective. Hence, we do not rewrite the collective but just call it through our hierarchy.

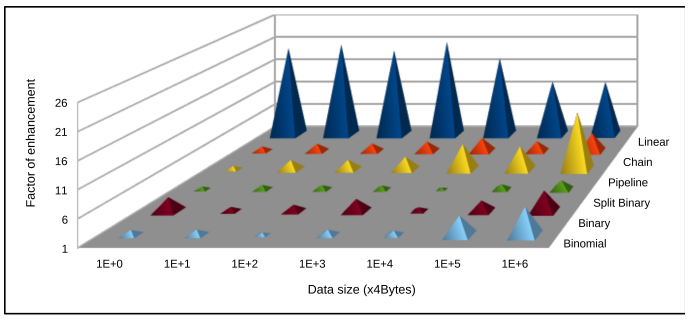

Figure 5: Enhancement factor of hierarchical approach for Open MPI Broadcast implementations on NTW10E5 (240cores)

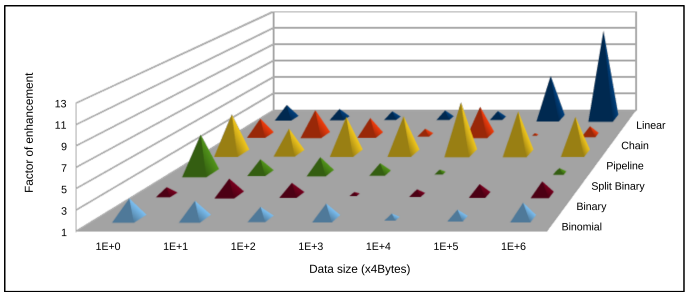

Figure 6: Enhancement factor of hierarchical approach for Open MPI Broadcast implementations SMP20E7 (160 cores)

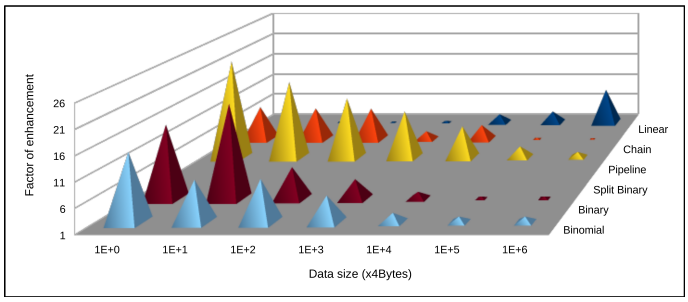

Figure 7: Enhancement factor of hierarchical approach for Open MPI Broadcast implementations SMP12E5 (96 cores)

Figures 5, 6 and 7 show the enhancement factor of the hierarchical-based implementation of six broadcast algorithms: Linear, Chain, Pipeline, Split Binary, Binary and Binomial. This factor is obtained by comparing the maximum time of processing several data sizes on the architectures described in Table 1 with the hierarchical broadcast and the native version of Open MPI broadcast. It is possible to note that the hierarchical approach we propose enhances almost all broadcast executions on the three architectures. In fact, the maxima achieved are roughly equal to 21x on NTW10E5, 11x on SMP20E5 and 22x on SMP12E5. This performance gain is due to two majors factors: first, our hierarchical approach allows to better exploit the parallelism and to reduce the complexity of broadcast algorithms. Second, the hardware topology-based communicators enhance the data locality and the hardware affinity between processes performing communications. Third, the communications are better pipelined over the network and the interconnect which reduce their total time.

Figures 8, 9 and 10 present the same results for the Reduce collective with six algorithms:

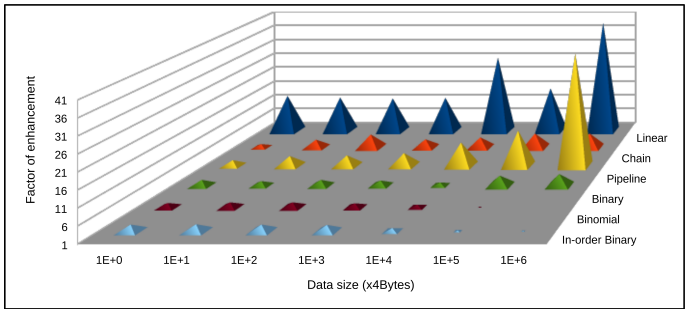

Figure 8: Enhancement factor of hierarchical approach for Open MPI Reduce implementations on NTW10E5 (240cores)

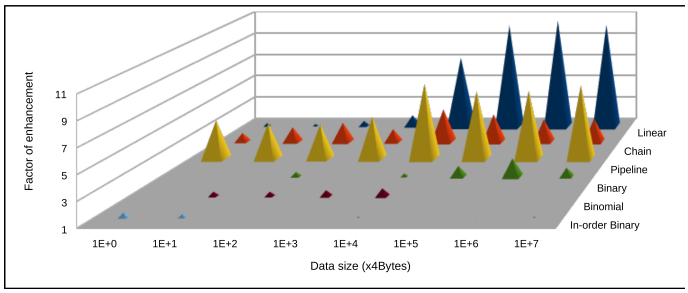

Figure 9: Enhancement factor of hierarchical approach for Open MPI Reduce implementations on SMP20E7 (160 cores)

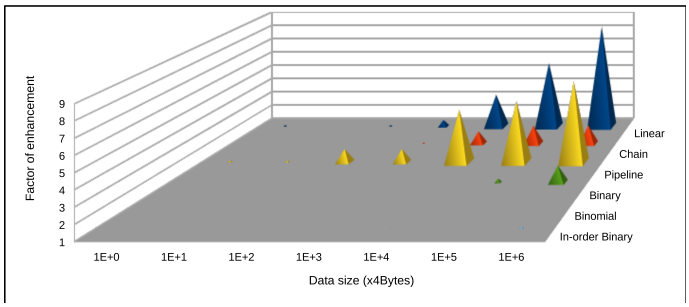

Figure 10: Enhancement factor of hierarchical approach for Open MPI Reduce implementations on SMP12E5 (96 cores)

Linear, Chain, Pipeline, Binary, Binomial and In-order Binary. In this case also the enhancement factor is obtained by comparing the maximum time for executing the reduction collective of several data sizes on the architectures described in Table 1 with both hierarchical and native Open MPI reduce. Here, we note that the hierarchical approach we propose considerably enhances the first three algorithms: Linear, Chain, Pipeline. In fact, the maxima achieved are roughly equal to 39x on NTW10E5, 11x on SMP20E5 and 8x on SMP12E5. These performances are achieved by the hardware-aware and hierarchical decomposition of the communications. Indeed, the algorithms are better parallelized, the data-locality is enhanced and the communications are better pipelined. However, we note that the three last algorithms (e.g. Binary, Binomial and In-order Binary) are less improved than the others if not at all. This phenomenon is due to the structure of the algorithms since they are already based on a hierarchical, tree-based structure. Therefore, the algorithm complexity is not enhanced by our approach. The small performances obtained for these algorithms are only due to the hardware optimisations and could be more significant with a larger

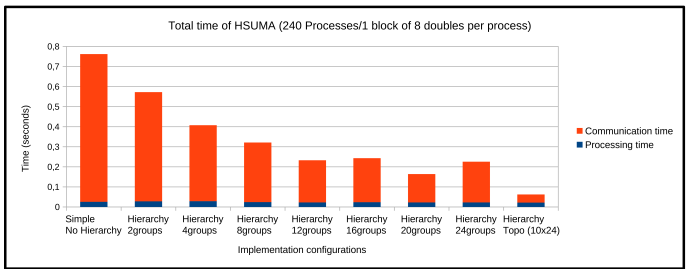

Figure 11: Comparison of execution time of simple, topology-oblivious hierarchical and topological matrix multiplication implementations on NTW10E5. 240 processes process 1 block of 8 doubles

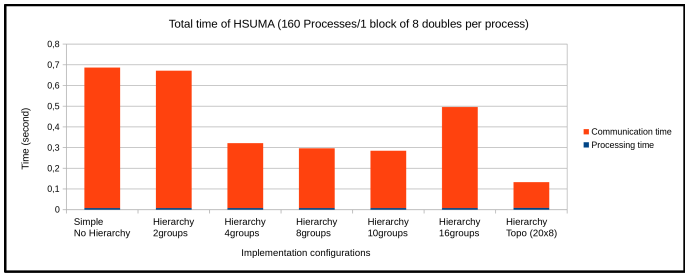

Figure 12: Comparison of execution time of simple, topology-oblivious hierarchical and topological matrix multiplication implementations on SMP20E7. 160 processes process 1 block of 8 doubles

number of processes.

7.3. Hierarchical Matrix Multiplication

The second benchmark we carried out is a two-dimensional, hierarchical matrix multiplication [23]. This application is a hierarchical extension of the SUMMA [24] algorithm based on two levels of hierarchy. Its implementation is hardware oblivious and is based on exploring several hierarchy configurations i.e. different number of communicators on each two levels. Hence, the programmer needs to manually specify the configuration at each execution by giving a configuration file as an input argument. With our approach, we abstract the manual specification of the hierarchy configuration. As a consequence we do not need to specify the configuration of the hierarchy in our implementation. It is based on the targeted architecture topology and automatically generated by using our proposed set of functions. The implementations we compare are:

- *Simple*: this is the implementation of the SUMMA algorithm based on broadcasting the blocks over rows and columns.
- *Hierarchical (xgroups)*: this is the hierarchical implementation of the SUMMA algorithm of matrix multiplication. The broadcasts of blocks are performed hierarchically over two levels. The hierarchy configuration is set by the user through a configuration file for each execution.
- *Hierarchical topological*: this is the same implementation but leveraging our communicators hierarchy. The hierarchy configuration is based on the underlying hardware topology and automatically build using our primitives.

Figures 11, 12 and 13 present the total processing times of several implementations: Simple (without hierarchy), Hierarchical with various configurations and Hierarchical with a topological

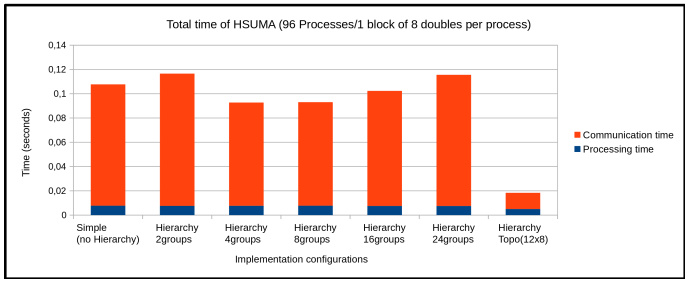

Figure 13: Comparison of execution time of simple, topology-oblivious hierarchical and topological matrix multiplication implementations on SMP12E5. 96 processes process 1 block of 8 doubles

configuration. The total time represents the needed time to process one block of 8 doubles per process and is composed of the average computing time represented by the blue part of the bar and the average communication time of processes by the red part. The used broadcast algorithm is the Open MPI implementation of Binary algorithm. We note that the Topological implementation (last bar) represents the minimum total time and achieve the better speed-ups of the simple implementation: 12x on NTW10E5, 5x on SMP20E7 and 6x on SMP12E5. Indeed, all implementations feature the same processing time (the blue part) but the communication time (the red part) is optimally reduced by the last implementation (Topological). This is due to the optimal hardware configuration matching with the topology. In fact, the communications inside hardware topology-aware communicators are more efficient thanks to the data locality and the MPI processes affinity. In addition, pipelining the communication enhances processors occupation time and reduces the global time.

7.4. Hierarchical complex vectors processing

The third benchmark we used in order to validate our approach is a simulation of hierarchical complex processing. This kind of application processes hierarchically a large set of data with different processing schemes at each level and for each process. Several Big Data domains such as hierarchical data labeling or hierarchic data hashing use the same idea to manage large sets of data. Algorithm 5 illustrates the functioning of this application where the users need to apply a function (`func`) on a local vector (`lvec`) as input argument on each rank to process gathered values from a ranks subset. In the implementation we use below, the function `func` is a simple vectors addition with normalization by value of a local vector `lvec`.

Algorithm 5:

```
1 rcomms ← root_based_hierarchic_communicators
2 func ← local_operation
3 lvec ← local_vector
4 foreach l← 0 .. (size_h) do // process in each level
5 |  MPI_Comm_rank(rcomms[l],&rank)
6 |  MPI_Gather(lresult[l], size_v, MPI_TYPE, temp[l], size_v, 0, rcomms[l])
7 |  if !rank then
8 |  |  lresult[l+1] ← func(temp[l],lvec)
```

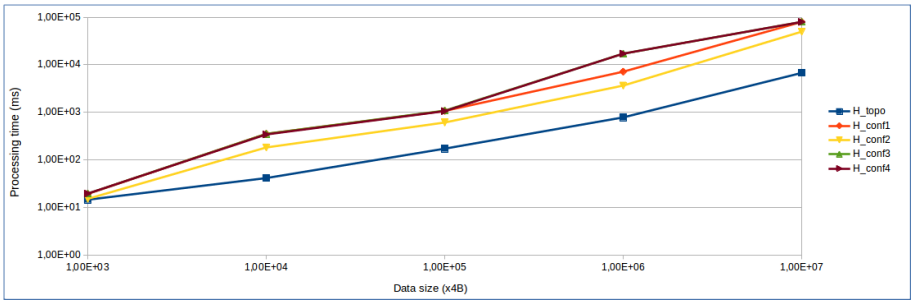

Figure 14: Comparison of execution times of topo-oblivious vs. topological implementations on NTW24E5. 576 processes process data size (x4B) with 5 hierarchical levels

We compare two implementations:

- H_topo: is an implementation based on our topology-aware communicator hierarchy where the communicators are generated by taking into account both the network and the node memory hierarchies. Therefore, in this implementation, the generated topology of communicator is based on the whole hardware topology including the number and levels of network switch, the number of compute nodes, the number of cores on each node and their memory hierarchy. For instance, in the experiment we carried out using NTW24E5, our topology of communicators contains in level number 1, 2, 3, 4 and 5 respectively: 1 communicator of 4 processes corresponding to the network switches, 4 communicators of various sizes (12, 6, 4 and 2 processes) corresponding to the physical nodes connected to each switch, 24 communicators of 2 processes corresponding to the sockets, 48 communicators of 2 processes corresponding to the NumaNodes and the last 96 communicators of 6 processes corresponding to the L2 cache level.

- H_conf(i): is the same implementation based on a tailor-made communicator hierarchy where the user manually build the hierarchy without taking into account the target underlying architecture. Since it is not easy to retrieve and understand the specifics of the target architecture topology, one can decide to use several possible configurations:

  Number of MPI process in communicator at level number 1, 2, 3, 4 and 5 :

  - H_conf1: 3, 4, 2, 3, 8
  - H_conf2: 6, 2, 2, 3, 8
  - H_conf3: 3, 2, 2, 8, 6
  - H_conf4: 3, 2, 4, 6, 4

Figure 14 shows the processing times (in log scale) of both implementations processing different data size. We can see that the implementation (blue curve) based on our hierarchy of communicators scales well and outperforms the rest of the other executions and reaches a speedup of about 10x. This is explained by the reduction of communication times (since the execution time is the same on each core because they process the same amount of data). In fact, thanks to our communicators topology we optimize the communication hierarchy by matching it with the hardware's. As opposed to this, the tailor-made communicator hierarchy is not hardware aware which implies no optimized

communications. Indeed, it is difficult for the user to know the specifics of the hardware when the resources are allocated by a batch scheduler and to build the communicator hierarchy accordingly. Our approach proposes a good abstraction to the MPI user to optimize its communications over a large and complex cluster.

8. Conclusion and Future Work

In this paper, we have presented an abstraction that can help the programmer to structure their application in order to take into account the hardware topology while designing it. We also presented how this model and abstraction could fit into an existing programming model/standard: the widely-used Message Passing Interface. One strength of our approach relies on its ability to address both network and nodes memory hierarchies at the same time. That is, if both cases are disjointed from an implementation point of view, they are not conceptually differently handled. Nodes memory hierarchy is seen as natural extensions of the network hierarchy. At the expense of light changes and a few new features introduction, we have shown that performance improvements can be achieved in MPI implementations themselves, but more importantly in parallel applications directly. Indeed, we introduced our topology-based communicators in two collective communication operations (Broadcast and Reduce) and showed that in the cases where hierarchy could improve performance, taking into account the physical topology improves things even further. Tested on four different architectures, our approach enhances the performance of the tested collective communications by factors up to 21x, 11x and 22x for the broadcast and 39x, 11x and 8x for the reduce. It also reduces the total time of hierarchical matrix multiplication with the factors of 12x, 5x and 6x. Experiments in which the network topology is also taken into account demonstrate very good results. Even if these gains of performance are considerable, the additional effort to use our approach is negligible and highly portable. In fact, thanks to the proposed abstraction automatically generating an hierarchy of communicators, the user does not have to deal with the details of hardware characteristics or manipulating low-level tools. He only deals with high-level MPI objects: communicators.

The hardware-agnosticism nature of MPI might seem paradoxical with our goal, as we precisely seek to offer the programmer means to better understand and exploit the underlying hardware. We believe that the independence from hardware considerations is an important strength of the MPI standard. We intend to keep MPI hardware-agnostic but we also believe that giving the programmer more hints about this same hardware can be very beneficial performance-wise. The predicament is therefore to find the relevant level of abstraction for such a new functionality. Indeed, if an MPI application should be able to gather and use specific pieces of information about the hardware, this information should nevertheless be abstract enough to not be tailored for a particular class of hardware. This work is available as an external library that features all the functions presented in this paper[7] We plan to further discuss this proposal at the MPI Forum in order to standardize it. We also plan to address non-hierarchical topologies, especially regarding the network. For instance, the graph representing a non-hierarchical network can always be partitioned (according to a criterion to define) and the split operation could then be performed accordingly. More generally, our proposal

[7]The code is available for download from `https://gforge.inria.fr/frs/download.php/file/36832/hsplit-rc-0.1.tar.gz` and from git at `https://scm.gforge.inria.fr/anonscm/git/mpi-topology/mpi-topology.git`

does not impose the underlying hardware to be hierarchically organized, as the same partitioning method can be employed to split the various application processes into the communicators. Also, expressing the topology with a distance function seems a promising idea and we would like to explore it.

Another future work deals with thread support. Currently our proposition relies on MPI processes making the relevant calls. However, it is possible with `hwloc` to retrieve information for each thread within a process. This issue is then to find the suitable MPI entity making the call to the split function. If MPI endpoints could be used to "represent" threads within a process, then this would be possible. However, this needs some clarifications regarding MPI processes and endpoints which is not in the scope of this paper nor this proposal.

Last, we believe that other objects in MPI implementations besides communicators could benefit from a knowledge of the underlying physical topology, for instance memory windows. Generalizing our approach could be of interest.

Acknowledgements

This work has partially been supported by the PIA ELCI project of the French FSN and by the ANR MOEBUS project ANR-13-INFR-0001.

Experiments presented in this paper were carried out using the PlaFRIM experimental testbed, being developed under the Inria PlaFRIM development action with support from Bordeaux INP, LABRI and IMB and other entities: Conseil Régional d'Aquitaine, Université de Bordeaux, CNRS and ANR in accordance to the programme d'Investissements d'Avenir (see `https://www.plafrim.fr/`).

The authors would like to thank the MPI Forum for its feedback, especially Daniel Holmes, as well as Cyril Bordage from implementing the `netloc` extension for retrieving network coordinates of nodes.

[1] D. E. Bernholdt, S. Boehm, G. Bosilca, M. G. Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, G. R. Vallee, A survey of mpi usage in the u. s. exascale computing project, in: EXAMPI workshop (in cunjunction with supercomuting), 2017, extended version submitted to Concurrency and Computation: Practice and Experience.

[2] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, R. Namyst, Hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications, in: Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010), IEEE Computer Society Press, Pisa, Italia, 2010.
URL `http://hal.inria.fr/inria-00429889`

[3] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, Version 3.0, Tech. rep. (September 2012).
URL `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`

[4] A. Kleen, A NUMA API for Linux, Novel Inc.
URL `http://halobates.de/numaapi3.pdf`

[5] B. Nichols, D. Buttlar, J. P. Farrell, Pthreads Programming, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[6] J. Reinders, J. Jeffers, High Performance Parallelism Pearls, 1st Edition, Vol. 2, Morgan Kaufmann, 2015.

[7] J. L. Träff, Implementing the MPI Process Topology Mechanism, in: Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, IEEE Computer Society Press, Los Alamitos, CA, USA, 2002, pp. 1–14.

[8] G. Mercier, E. Jeannot, Improving MPI Applications Performance on Multicore Clusters with Rank Reordering, in: EuroMPI, Vol. 6960 of Lecture Notes in Computer Science, Springer, Santorini, Greece, 2011, pp. 39–49.

[9] M. J. Rashti, J. Green, P. Balaji, A. Afsahi, W. Gropp, Multi-core and Network Aware MPI Topology Functions, in: Y. Cotronis, A. Danalis, D. S. Nikolopoulos, J. Dongarra (Eds.), EuroMPI 2011. Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, Vol. 6960 of Lecture Notes in Computer Science, Springer, 2011, pp. 50–60.

[10] E. Jeannot, G. Mercier, F. Tessier, Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques, IEEE Trans. Parallel Distrib. Syst. 25 (4) (2014) 993–1002. doi:10.1109/TPDS.2013.104.
URL http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.104

[11] T. Hatazaki, Rank Reordering Strategy for MPI Topology Creation Functions, in: V. Alexandrov, J. Dongarra (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, Vol. 1497 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 1998, pp. 188–195, 10.1007/BFb0056575.
URL http://dx.doi.org/10.1007/BFb0056575

[12] Jesper Larsson Träff, Implementing the MPI process topology mechanism, in: Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, IEEE Computer Society Press, Los Alamitos, CA, USA, 2002, pp. 1–14.

[13] J. Hursey, J. M. Squyres, T. Dontje, Locality-Aware Parallel Process Mapping for Multi-core HPC Systems, in: 2011 IEEE International Conference on Cluster Computing (CLUSTER), IEEE, 2011, pp. 527–531.

[14] G. Mercier, J. Clet-Ortega, Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments, in: EuroPVM/MPI, Vol. 5759 of Lecture Notes in Computer Science, Springer, Espoo, Finland, 2009, pp. 104–115.

[15] B. Brandfass, T. Alrutz, T. Gerhold, Rank Reordering for MPI Communication Optimization, Computer & Fluidsdoi:http://dx.doi.org/10.1016/j.compfluid.2012.01.019.

[16] J. Zhang, J. Zhai, W. Chen, W. Zheng, Process Mapping for MPI Collective Communications, in: H. J. Sips, D. H. J. Epema, H.-X. Lin (Eds.), Euro-Par, Vol. 5704 of Lecture Notes in Computer Science, Springer, 2009, pp. 81–92.

[17] B. Goglin, J. Hursey, J. M. Squyres, Netloc: Towards a comprehensive view of the HPC system topology, in: 43rd International Conference on Parallel Processing Workshops, ICPPW 2014, Minneapolis, MN, USA, September 9-12, 2014, IEEE Computer Society, 2014, pp. 216–225.

doi:10.1109/ICPPW.2014.38.
URL https://doi.org/10.1109/ICPPW.2014.38

[18] R. W. Hockney, The communication challenge for mpp: Intel paragon and meiko cs-2, Parallel Comput. 20 (3) (1994) 389–398. doi:10.1016/S0167-8191(06)80021-9.
URL http://dx.doi.org/10.1016/S0167-8191(06)80021-9

[19] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. von Eicken, Logp: Towards a realistic model of parallel computation, SIGPLAN Not. 28 (7) (1993) 1–12. doi:10.1145/173284.155333.
URL http://doi.acm.org/10.1145/173284.155333

[20] T. Kielmann, H. E. Bal, K. Verstoep, Fast Measurement of LogP Parameters for Message Passing Platforms, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 1176–1183. doi:10.1007/3-540-45591-4_162.
URL http://dx.doi.org/10.1007/3-540-45591-4_162

[21] Plafrim, Plate-forme fédérative pour la recherche en informatique et mathématiques, https://plafrim.bordeaux.inria.fr/doku.php.

[22] H. Zhu, D. Goodell, W. Gropp, R. Thakur, Hierarchical Collectives in MPICH2, in: Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 325–326.

[23] J.-N. Quintin, K. Hasanov, A. Lastovetsky, Hierarchical parallel matrix multiplication on large-scale distributed memory platforms, in: 42nd International Conference on Parallel Processing (ICPP 2013), IEEE, IEEE, Lyon, France, 2013, pp. 754–762. doi:DOI 10.1109/ICPP.2013.89.

[24] R. A. Van De Geijn, J. Watts, Summa: scalable universal matrix multiplication algorithm, Concurrency: Practice and Experience 9 (4) (1997) 255–274. doi:10.1002/(SICI)1096-9128(199704)9:4<255::AID-CPE250>3.0.CO;2-2.
URL http://dx.doi.org/10.1002/(SICI)1096-9128(199704)9:4<255::AID-CPE250>3.0.CO;2-2